.МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ
«ХАРКІВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ»

МЕТОДИЧНІ ВКАЗІВКИ

до лабораторних робіт

«Використання SIMD розширення обчислювача

при програмуванні в середовищі VISUAL STUDIO»

з дисципліни «Архітектура обчислювальних систем»

для студентів спеціальностей

113 – Прикладна математика, 122 – Комп'ютерні науки та інформаційні технології, 124 – Системний аналіз, 186 – Видавництво та поліграфія

Затверджено
редакційно-видавничою
радою університету,
протокол № 1 від 22.06.17

Харків
НТУ «ХПІ»
2017

Методичні вказівки до лабораторних робіт «Використання SIMD розширення обчислювача при програмуванні в середовищі VISUAL STUDIO» з дисципліни «Архітектура обчислювальних систем» : для студентів спеціальностей 113 – Прикладна математика, 122 – Комп'ютерні науки та інформаційні технології, 124 – Системний аналіз, 186 – Видавництво та поліграфія / уклад.: Ю. М. Кожин, О. М. Малих, В. П. Прокопенков. – Харків : НТУ «ХПІ», 2017. – 40 с.

Укладачі: Ю. М. Кожин
О. М. Малих
В. П. Прокопенков

Рецензент Н. І. Безменов

Кафедра системного аналізу та інформаційно-аналітичних технологій

Вступ

Характерною рисою сучасних систем управління є вимога до забезпечення функціонування в реальному масштабі часу. Забезпечити цю вимогу можливо завдяки застосуванню паралельних і матричних процесорів з одним потоком команд і багатьма потоками даних (*SIMD – single instruction, multiple data*).

Основна ідея *SIMD* обчислень полягає в одночасному обробленні декількох елементів даних (векторів) за одну команду.

У систему команд процесора *INTEL* уведені додаткові *SSE* операції обробки векторних даних. Застосування таких операцій при обробленні масивів даних дозволяє значно скоротити період обчислень. У той же час розробка програмного забезпечення з використанням *SIMD* обчислень призводить до збільшення часу на проектування програм.

Метою лабораторних робіт є освоєння прийомів аналізу архітектури обчислювача для використання *SSE*-команд, застосування асемблера для розробки програм обробки векторних даних, а також отримання навичок проектування додатків операційної системи Windows для організації SIMD-обчислень.

1. ПЕРЕВІРКА ІСНУВАННЯ SIMD-РОЗШИРЕННЯ ОБЧИСЛЮВАЧА

Мета роботи: навчитися аналізувати наявність SIMD розширення архітектури x86 при програмуванні в операційній системі Windows.

1.1. SIMD-розширення архітектури x86

SIMD-розширення (*Single Instruction Multiple Data*) були введені в архітектуру x86 з метою підвищення швидкості обробки потокових даних. Основна ідея полягає в одночасному обробленні декількох елементів даних за одну інструкцію. Програмна модель процесора з урахуванням *SIMD*-розширення подана на рис. 1.1.

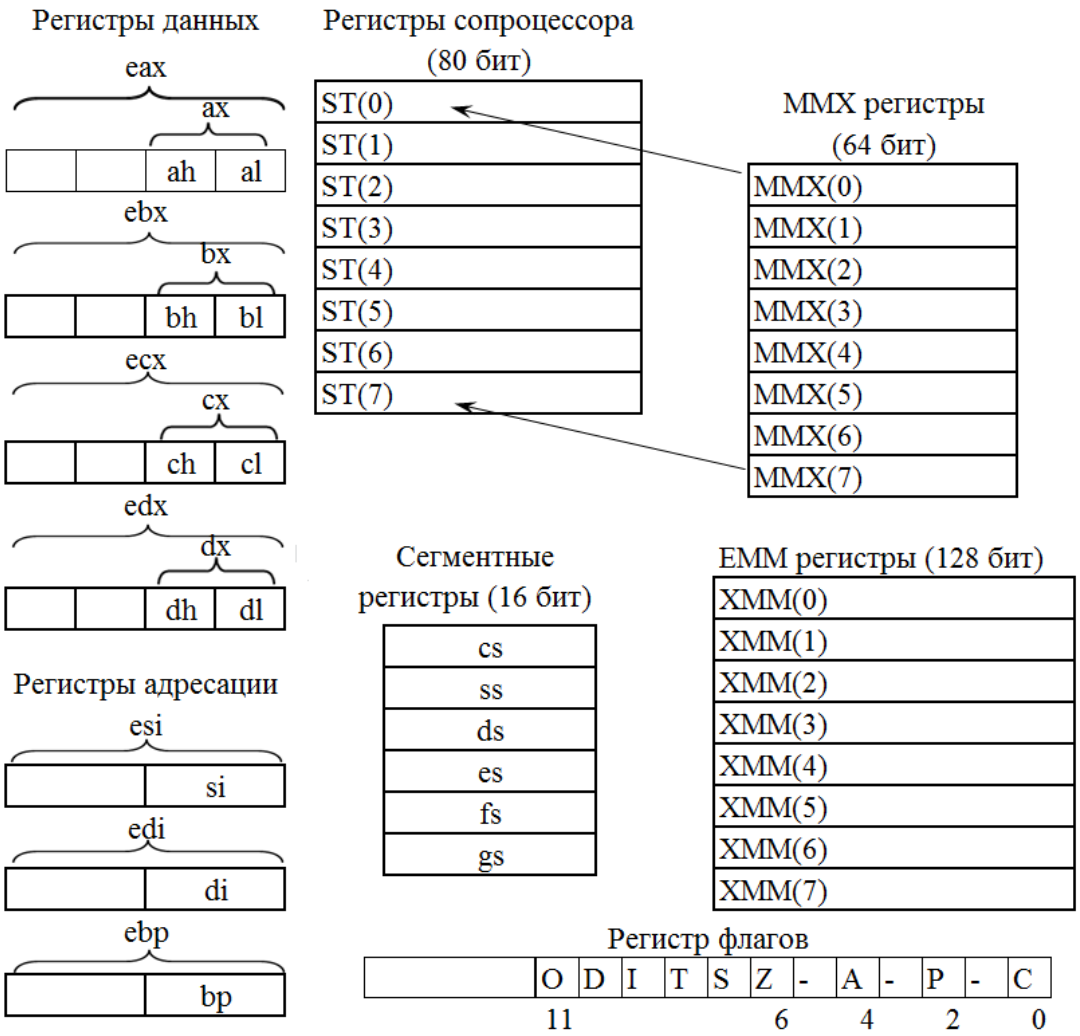

Рисунок 1.1 – Програмна модель процесора

Регістри загального призначення *EAX, EBX, ECX, EDX* мають розрядність 32 біта і використовуються для обробки цілих чисел. Адресні регістри *ESI, EDI, EBP, ESP* також мають розрядність 32 біта і застосовуються для різних способів адресації до пам'яті обчислювача.

Регістри співпроцесора *ST* (0) – *ST* (7) служать для виконання операцій з дійсними даними. Регістри мають розрядність 80 біт і дозволяють обробляти числові дані з плаваючою комою у форматі одинарної точності (*float*), подвійної точності (*double*) і розширеної точності (*extended*).

Регістри *MMX* мають розрядність 64 біта. Регістри *MMX* фізично реалізовані як складова частина регістрів співпроцесора. При цьому використовується тільки 64 розряди відповідного регістра співпроцесора.

Регістри *MMX* беруть участь в обробленні цілочисельних даних. Особливістю регістрів *MMX* є те, що кожен регістр може зберігати вісім однобайтових або чотири двохбайтових або два чотирьохбайтових цілочисельних значень, або одно 64-розрядне ціле число. Одна і та ж операція може бути виконана над усіма однотипними даними регістра *MMX* (SIMD-обчислення – одна інструкція, багато даних).

З процесором *Intel Pentium* III уперше з'явилося розширення *SSE* (*Streaming SIMD Extension*). Це розширення працює з блоком з восьми 128-бітових регістрів *XMM0 – XMM7*. Кожен регістр *XMM* може обробляти чотири упакованих 32-бітових одинарної точності або два 64-бітових подвійної точності дійсних числа. Команди блоку *XMM* дозволяють виконувати як векторні (над усіма чотирма або двома значеннями регістра), так і скалярні операції (тільки над одним наймолодшим значенням). Окрім інструкцій з блоком *XMM* в розширення *SSE* входять і додаткові інструкції з регістрами *MMX*.

1.2. Ідентифікація устаткування і програмного оточення

1.2.1. Команда визначення параметрів процесора

Для визначення параметрів процесора та наявність *MMX* та *SSE* розширень використовується команда асемблера *cpuid*, яка має аналог на мові *C/C++* _ _*cpuid*( ).

Так, для визначення властивостей процесора на мові *C++* можна використати такий код:

```
#include <intrin.h>          // підключення опису функції _ _cpuid
int CPU _ Info[4];
char CPU_S[32];
_ _cpuid (CPUInfo, 0);
memset (CPU_S, 0, sizeof (CPU_S));
*((int*)CPU_S)       = CPU _ Info[1];
*((int*)(CPU_S+4)) = CPU _ Info[3];
*((int*)(CPU_S+8)) = CPU _ Info[2];
```

Перший параметр функції _ _*cpuid* – 4-елементний цілочисельний масив, який відповідає регістрам *eax, ebx, ecx, edx* центрального процесора після виконання інструкції. Другий параметр – тип інформації про процесор.

З використанням асемблера інформацію про процесор можна отримати послідовністю команд:

```
MOV EAX, 0
CPUID
```

Для різних моделей процесорів існують різні набори припустимих вхідних значень. У загальному випадку їх можна розділити на стандартні (підтримувані усіма виробниками) та розширені (такими, що відрізняються для процесорів різних моделей і виробників).

1.2.2. Ідентифікація виробника процесора

При значенні *EAX* рівним нулю після команди *CPUID* в *EAX* міститиметься максимальне значення, яке можна використати у команді *CPUID* (тільки стандартні функції), а в регістрах *EBX, ECX* і *EDX* знаходитиметься рядок ідентифікації виробника процесора (по чотири символи в кожному регістрі). У табл. 1.1 наведені значення, що видаються найбільш поширеними моделями мікропроцесорів.

Таблиця 1.1 – Рядок ідентифікації виробника

Тип процесора (виробник)	*EAX*	*EBX ECX EDX*
486SL … Pentium III Xeon (Intel)	X	GenuineIntel
6x86(L) (Cyrix)	1	CyrixInstead
MediaGX MMX Enhansed (Cyrix)	2	CyrixInstead
WinChip 2 (IDT)	1	CentaurHauls
AMD–K6–3 (AMD)	1	AuthenticAMD

1.2.3. Ідентифікація моделі процесора

Якщо *EAX* до команди *CPUID* містить одиничне значення, то окремі розряди *EAX* після виконання команди міститимуть таку інформацію:

[3: 0] – (*Stepping*) номер розробки мікропроцесора;

[7: 4] – (*Model*) модель мікропроцесора;

[11: 8] – (*Family*) номер сімейства мікропроцесора;

[13: 12] – (*Type*) тип мікропроцесора (00b – стандартний, 01b – *OverDrive*, 10b – *Dual*;

[31: 14] – зарезервовано.

У *EDX* міститиметься інформація про деякі властивості і можливості мікропроцесора. Встановлений прапор властивостей вказує на те, що відповідна властивість (можливість, функція) цією моделлю мікропроцесора підтримується. У табл. 1.2 подані прапори властивостей процесора.

Таблиця 1.2 – Значення біт властивостей процесора

Біт *EDX*	Опис
0	Наявність *FPU*, що підтримує систему команд *Intel387*
4	Підтримка команди *RDTSC*
5	Підтримка команди *RDMSR* та *WRMSR*
6	Підтримка розширень фізичної адреси (більше 4 Гб)
8	Підтримка команди *CMPXCHG8B*;
18	Підтримка унікального 96-розрядного серійного номера
23	Підтримка команд *MMX*
24	Підтримка команд швидкого збереження/відновлення стану *FPU/MMX/SIMD* (*FXSAVE, FXRSTOR*)
25	Підтримка команд *SIMD SSE*
26	Підтримка команд *SIMD SSE*2
28	Підтримка *Hyper Threading*

1.2.4. Визначення об'єму та типу кеш пам'яті процесора

*Я*кщо *EAX* має значення 2, команда *CPUID* дозволяє отримати відомості про об'єм і тип кеш пам'яті мікропроцесора. Після виконання команди в *EAX, EBX, EDX, ECX* міститься відповідна інформація, причому у молодших восьми бітах регістра *EAX* (регістр *AL*) міститься число, що вказує на те, скільки разів підряд необхідно виконати команду *CPUID* з вхідним *EAX* = 2, щоб отримати достовірну інформацію про мікропроцесор (у разі *AL* = 1, команда повинна виконуватися одноразово).

Самий старший біт в кожному 32-бітовому регістрі визначає достовірність інформації (біт 31 = 0) або резервування (біт 31 = 1). Регістри *EAX, EBX, EDX, ECX* містять дескриптори, що описують кеш пам'ять процесора. Деякі значення цих дескрипторів подані в табл. 1.3.

Таблиця 1.3 – Дескриптори опису кеш пам’яті процесора

Значення	Опис
00h	Нуль-дескриптор (без значення)
01h	*TLB* команд, 4К сторінки: 4-спрямований асоціативний, 32 елементи
02h	*TLB* команд, 4М сторінки: повністю асоціативний, 2 елементи
03h	*TLB* даних, 4К сторінки: 4-спрямований асоціативний, 64 елементи
04h	*TLB* даних, 4М сторінки: 4-спрямований асоціативний, 8 елементів
06h	Кеш команд: розмір 8К, 4-спрямований асоціативний, рядки 32 байти
0Ah	Кеш даних: розмір 8К, 2-спрямований асоціативний, рядки 32 байти
0Ch	Кеш даних: розмір 16К, 4-спрямований асоціативний, рядки 32 байти
40h	Інтегрований кеш другого рівня відсутній (*L2 cache*)
41h	Об’єднаний кеш: раз розмір мер 128К, 4-спрямований асоціативний, рядки 32 байти
42h	Об’єднаний кеш: розмір 256К, 4-спрямований асоціативний, строки 32 байти
43h	Об’єднаний кеш: розмір 512К, 4-спрямований асоціативний, рядки 32 байти
44h	Об’єднаний кеш: розмір 1М, 4-спрямований асоціативний, довжина 32 байти
45h	Об’єднаний кеш: розмір 2М, 4-спрямований асоціативний, довжина рядки 32 байти

1.2.5. Отримання ідентифікаційного номера процесора

Команда *CPUID* з вхідним значенням *EAX* = 3 призначена для отримання ідентифікаційного номера процесора – унікального 96-розрядного номера. Після виконання цієї команди регістр *ECX* містить молодші 32 біта ідентифікаційного номера, а регістр *EDX* – середні 32 біта.

Старші 32 біта ідентифікаційного номера видаються в регістрі *EAX* командою *CPUID* з вхідним *EDX* = 1. Стандартна форма запису ідентифікаційного номера, визначувана фірмою Intel, виглядає таким чином: XXXX - XXXX - XXXX - XXXX - XXXX (X – шістнадцятирічна цифра).

1.2.6. Перевірка підтримки виконання команди CPUID

Для того щоб дізнатися, чи підтримується команда *CPUID* конкретною моделлю процесора, потрібно перевірити біт за номером 21 (*ID*) регістра *EF*. Якщо у програмі можна встановити та скинути цей прапор, то це означає команда *CPUID* цим процесором підтримується. Інакше, виконання цієї команди може викликати виключення.

Нижче наведений приклад програми, що визначає підтримку команди *CPUID*.

```
pushfd ;            EFLAGS копіювати в стек
pop eax ;           вибрати в EAX вміст стека
mov ebx, eax ;      зберегти вибране значення прапорів
xor eax, 200000h ; змінити біт 21 у вибраному значенні прапорів
push eax ;          помістити нове значення прапорів у стек
popfd ;             записати в EFLAGS із стека нові значення прапорів
pushfd ;            EFLAGS копіювати в стек
pop eax ;
xor eax, ebx ;      порівняти нове значення прапорів
je no_cpuid ;       перехід, якщо команда cpuid не підтримується
```

1.3. Функції виміру часу

Сучасні процесори можуть мати спеціальний регістр – лічильник міток реального часу (Time Stamp Counter), вміст якого збільшується на одиницю з кожним тактом процесорного ядра. Цей регістр може бути використаний для оцінення продуктивності процесора та/або для контролю часу виконання ділянки коду програми. Для доступу до вмісту регістра використовується інструкція процесора *rdtsc*.

Коли прапор *TSD* регістра *CR4*, що управляє, встановлений, команда rdtsc може виконуватися тільки в захищеному режимі при рівні привілеїв, рівному 0, коли ж цей прапор скинутий, команда *rdtsc* перестає бути привілейованою і стає доступною в усіх режимах.

Перед виконанням команди rdtsc необхідно переконатися в тому, що цей процесор і його режим роботи дозволяє виконувати команду *rdtsc*.

Часто для виміру часу застосовують функції *API QueryPerformanceFrequency* і *QueryPerformanceCounter*, які самі здійснюють необхідні перевірки. Функція *QueryPerformanceFrequency* дозволяє отримати число тактів процесора за одну секунду (тактова частота процесора), а *QueryPerformanceCounter* – поточне значення лічильника.

Приклад використання цих функцій наведений нижче:

```
#include <stdio.h>
#include <windows.h>
int main()
{       LARGE_INTEGER b_start,b_stop,b_time,freq;
        double time, p;
        QueryPerformanceFrequency(&freq);
        QueryPerformanceCounter(&b_start);
              p = calculate( );
        QueryPerformanceCounter(&b_stop);
        b_time.QuadPart = b_stop.QuadPart – b_start.QuadPart;
```

```
    printf("Time: %lf sec P = %lf\n",
        (double) (b_time.QuadPart)/ (double) (freq.QuadPart) ,pi);
    return 0;
}
```

У прикладі визначається час виконання функції *calculate*( ).

1.4. Порядок виконання роботи

1. Розробити програму консольного типу з використанням модулів асемблера. Основний модуль реалізувати на *C/C++* . Модуль забезпечує діалог з користувачем та виведення даних. Модуль асемблера забезпечує отримання необхідних даних про процесор відповідно до варіанта.

2. Перевірити роботу програми на різних обчислювачах. Звірити отримані відомості з відомостями від операційної системи.

3. Оформити звіт.

Варіанти завдань:

1. Перевірити, чи є поточний процесор мікропроцесором фірми *INTEL*. Отримати номер сімейства мікропроцесора і модель. Визначити наявність інтегрованого *FPU*.

2. Перевірити, чи є поточний мікропроцесор процесором фірми *AMD*. Отримати номер розробки і тип мікропроцесора. Визначити підтримку команд роботи з лічильником міток реального часу.

3. Ідентифікувати виробника мікропроцесора, номер розробки і модель мікропроцесора. Перевірити, чи є цей мікропроцесор стандартним і чи підтримує він режим *Hyper Threading*.

4. Перевірити, чи є поточний мікропроцесор процесором фірми *INTEL*. Отримати номер сімейства мікропроцесора і модель. Перевірити, чи є підтримка розширень фізичної адреси.

5. Перевірити чи є поточний мікропроцесор процесором фірми *AMD*. Отримати номер розробки і тип мікропроцесора. Перевірити підтримку команд швидкого збереження/відновлення стану *FPU/MMX/SIMD*.

6. Ідентифікувати виробника мікропроцесора, номер розробки і модель мікропроцесора. Перевірити чи є цей мікропроцесор стандартним. Отримати унікальний 96-розрядний серійний номер.

Контрольні питання

1. Для чого використовуються регістри співпроцесора *ST* (0) – *ST* (7) і яка їх розрядність?
2. Для чого використовуються регістри *MMX*-розширення і яка їх розрядність?
3. Які типи даних дозволяють обробляти *MMX*-регістри?
4. Якою є розрядність регістрів *SSE*-розширення і для чого воні використовуються?
5. Які типи даних дозволяють обробляти *XMM*-регістри?
6. У якому випадку слід використовувати команду асемблера *cpuid*?
7. Як здійснити ідентифікацію виробника процесора?
8. У яких розрядах акумулятора зберігається модель і номер сімейства процесора?
9. Скільки різних режимів має команда *cpuid*?
10. Що таке ідентифікаційний номер мікропроцесора та як його отримати?
11. Як здійснюється перевірка можливості виконання команди *cpuid*?

2. ВИКОРИСТАННЯ SSE КОМАНД АСЕМБЛЕРА

Мета роботи: вивчення системи команд *SIMD* розширення архітектури x86 для розробки програм в операційній системі Windows.

2.1. Апаратні розширення процесора

2.1.1. Розширення MMX

Вперше *SIMD*-розширення ввела фірма *Intel* у мікропроцесорах Pentium *MMX* (розширення архітектури *Pentium* або P5) і *Pentium* II (розширення архітектури *Pentium Pro* або P6). Розширення *MMX* працює з 64-бітовими регістрами *MM*0 – *MM*7, фізично розташованими на регістрах співпроцесора, і включає 57 нових інструкцій для роботи з ними. 64-бітові регістри можуть представлятися як одно 64-бітове, два 32-бітових, чотири 16-бітових або вісім 8-бітових упакованих цілих. Ще одна особливість технології MMX – арифметика з насиченням, в якій при переповнюванні фіксується мінімальне або максимальне значення. Наприклад, для 8-бітового цілого без знаку:

звичайна арифметика– x=254; x+=3; // результат x= 1

арифметика з насиченням– x=254; x+=3; // результат x=255

2.1.2. Розширення SSE2

У процесорі *Intel Pentium* 4 набір інструкцій отримав чергове розширення – *SSE*2. Воно дозволяє працювати з 128-бітовими регістрами *XMM* як з парою упакованих 64-бітових дійсних чисел подвійної точності, а також з упакованими цілими числами: 16 байт, 8 слів, 4 подвійні слова або 2 збільшені вчетверо слова (64-бітови). Введені нові інструкції дійсної арифметики подвійної точності, інструкції цілочисельної арифметики, 128-розрядні для регістрів *XMM* та 64-розрядні для регістрів *MMX*. Інструкції *MMX* розповсюдили і на *XMM* (у 128-бітовому варіанті).

2.1.3. Розширення SSE3

Подальше розширення системи команд *SSE*3 вводиться у мікропроцесорі *Intel Pentium* 4 з ядром *Prescott*. Це набір з 13 нових інструкцій, працюю-

чих з блоками *XMM, FPU*, у тому числі двох інструкцій, що підвищують ефективність синхронізації потоків, зокрема, при використанні технології *HyperThreading*.

2.1.4. Підтримка SIMD-розширень архітектурою x86 – 64

Процесори *AMD Athlon*64 і *AMD Opteron* з архітектурою x86-64 підтримують все перелічені вище *SIMD*-розширення, окрім *SSE*3. Крім того, число *XMM* регістрів у цих мікропроцесорах збільшилося до 16 (*XMM*0 – *XMM*15).

2.2. Типи даних

Для роботи з векторними даними, що містять декілька упакованих значень, використовуються такі типи:

2 × 32-бітових дійсних (одинарна точність);

2 × 64-бітових дійсних (подвійна точність);

2 × 64-бітових цілих;

4 × 32-бітових цілих;

8 × 16-бітових цілих;

16 × 8-бітових цілих.

2.3. Система команд *SSE* розширення

2.3.1. Скалярні і векторні операції

У системі команд *SSE* є скалярні і векторні операції. У скалярній операції беруть участь тільки молодші частини операндів, у векторних – усі елементи.

Наприклад, скалярне складання чотирьох дійсних 32-розрядних чисел виконуватиметься, як показано на рис.2.1, а векторне складання – як на рис. 2.2.

	Четвертий елемент	Третій елемент	Другий елемент	Перший елемент
Перший операнд	21.0	34.0	78.0	23.0
Операція				+
Другий операнд	45.0	88.0	4.0	12.0
Результат	21.0	34.0	78.0	35.0

Рисунок 2.1 – Скалярне складання двох операндів

	Четвертий елемент	Третій елемент	Другий елемент	Перший елемент
Перший операнд	21.0	34.0	78.0	23.0
Операція	+	+	+	+
Другий операнд	45.0	88.0	4.0	12.0
Результат	56.0	122.0	82.0	35.0

Рисунок 2.2 – Векторне складання двох операндів

2.3.2. Арифметичні операції

Система команд *SSE*2 підтримує арифметичні операції складання, віднімання, множення та ділення дійсних чисел.

У арифметичних командах використовуються два операнди. Перший операнд має бути *XMM* регістром, другим операндом може виступати регістр або елемент пам’яті відповідного розміру. Для вказівки як другий операнд елемента пам’яті може бути використаний будь-який режим адресації. Результат виконання операції поміщається в перший операнд.

Арифметичні команди мають синтаксис:

OP RXMM, RXMM / MEM ,

де *OP* – код арифметичної операції, *RXMM* – регістр *XMM*, *MEM* – елемент пам’яті. У табл. 2.1 наведені мнемоніки арифметичних операцій *SSE2*.

Таблиця 2.1 – Арифметичні операції *SSE2*

Операція	Мнемоніка	Кількість елементів	Розрядність елементів
	Векторні		
Складання	*ADDPD*	2	64
	ADDPS	4	32
Віднімання	*SUBPD*	2	64
	SUBPS	4	32
Множення	*MULPD*	2	64
	MULPS	4	32
Ділення	*DIVPD*	2	64
	DIVPS	4	32
	Скалярні		
Складання	*ADDSD*	1	64
	ADDSS	1	32
Віднімання	*SUBSD*	1	64
	SUBSS	1	32
Множення	*MULSD*	1	64
	MULSS	1	32
Ділення	*DIVSD*	1	64
	DIVSS	1	32

При виконанні скалярних операцій старші елементи регістра приймача не змінюються.

2.3.3. Операції обчислення кореня та групові операції

Система команд *SSE*2 дозволяє виконувати операції обчислення квадратного кореня, обчислення зворотного значення, зворотного значення квадратного кореня, знаходження мінімального і максимального чисел.

В операції беруть участь два операнди: перший – приймач результату, другий – джерело даних. Джерелом даних може виступати елемент пам'яті з будь-яким режимом адресації або регістр *XMM*, приймачем даних завжди є регістр *XMM*. Команда обчислення кореня має синтаксис:

OP RXMM, RXMM / MEM ,

де *OP* – код операції, *RXMM* – регістр *XMM, MEM* – елемент пам'яті.

При знаходженні мінімального і максимального значень у відповідний елемент приймача записується мінімум або максимум між значеннями елементів джерела і приймача.

У табл. 2.2 показані команди обчислення квадратного кореня і зворотних значень.

Таблиця 2.2 – Команди обчислення квадратного кореня і зворотних значень

Операція	Мнемоніка	Кількість елементів	Розрядність елементів
	Векторні		
Корінь квадратний	*SQRTPD*	2	64
	SQRTPS	4	32
Зворотне значення	*RCPPS*	4	32
Зворотне значення кореня	*RSQRTPS*	4	32
Максимум	*MAXPD*	2	64
	MAXPS	4	32

Продовження табл. 2.2

Операція	Мнемоніка	Кількість елементів	Розрядність елементів
Мінімум	*MINPD*	2	64
	MINPS	4	32
Корінь квадратний	*SQRTSD*	1	64
	SQRTSS	1	32
	Скалярні		
Зворотне значення	*RCPSS*	1	32
Зворотне значення кореня	*RSQRTSS*	1	32
Максимум	*MAXSD*	1	64
	MAXSS	1	32
Мінімум	*MINSD*	1	64
	MINSS	1	32

У операції переміщення є два операнди: перший операнд приймач, другий – джерело даних. Приймачем і джерелом може виступати регістр *XMM* або елемент пам’яті відповідного розміру. Забороняється вказувати одночасно як джерело, так і приймач даних елемент пам’яті. Команда переміщення має синтаксис:

MOVXXX RXMM / MEM, RXMM / MEM,

де *MOVXXX* – код операції, *RXMM* – регістр *XMM*, *MEM* – елемент пам’яті.

У скалярних операціях, якщо як джерела пересилки даних вказується елемент пам’яті, то старші елементи приймача дорівнюватимуть нулю.

Бажано щоб адреса елемента пам’яті при векторному зчитуванні або запису даних в оперативну пам’ять (ОЗУ) була кратна 16. Якщо дані "вирівняні", то звернення до ОЗУ займає 1 такт, інакше 2 такти. Для вирівнювання адреси розміщення статичних даних слід використати директиву ALIGN 16

У табл. 2.3 подані команди передачі даних.

2.3.4. Операції порівняння даних

У операції порівняння беруть участь три операнди: перші два – порівнювані дані, третій – константа, що вказує умову порівняння. Команди порівняння мають синтаксис:

Таблиця 2.3 – Команди переміщення значень

Операція	Мнемоніка	Кількість елементів	Розрядність елементів
	Векторні		
Вирівняні дані	*MOVAPD*	2	64
	MOVAPS	4	32
Не вирівняні дані	*MOVUPD*	2	64
	MOVUPS	4	32
	Скалярні		
Подвійна точність	*MOVSD*	1	64
Одинарна точність	*MOVSS*	1	32

CMPPD RXMM, RXMM / MEM, CONST

CMPPS RXMM, RXMM / MEM, CONST

CMPSD RXMM, RXMM / MEM, CONST

CMPSS RXMM, RXMM / MEM, CONST,

де *RXMM* – регістр *XMM, MEM* – елемент пам’яті.

В табл. 2.4 наведені команди порівняння *SSE*.

Константа може набувати значення від 0 до 7 відповідно до умов порівняння (табл. 2.5).

Таблиця 2.4 –Команди порівняння

Команда	Кількість елементів	Розрядність елементів
Векторні		
CMPPD	2	64
CMPPS	4	32
Скалярні		
CMPSD	1	64
CMPSS	1	32

Таблиця 2.5 – Коди константи в операції порівняння

Код константи	Операція порівняння	Примітка
0	Приймач = = Джерело	Рівність значень
1	Приймач < Джерело	Менше
2	Приймач ≤ Джерело	Менше або рівно
3	Приймач і/або Джерело не можна порівнювати	Не можна порівнювати
4	Приймач ≠ Джерело	Не рівно
5	Не (Приймач < Джерело)	Не менше
6	Не (Приймач ≤ Джерело)	Більше
7	Приймач і Джерело можна порівнювати	Можна порівнювати

Для векторних операцій у приймач результату у відповідний елемент записується нульове значення, якщо умова не виконується і усі одиниці, якщо умова виконується.

У скалярних операціях беруть участь тільки молодші елементи. Старші елементи приймача не змінюються.

2.3.5. Збереження та відновлення регістрів розширень обчислювача

Для організації роботи з підпрограмами в системі команд *SSE*2 передбачено збереження регістрів співпроцесора, регістрів *MMX* та *XMM* розширень. Для збереження даних потрібно 512 байт пам'яті. Адреса області пам'яті для збереження має бути вирівняна на межу кратну 16.

```
.DATA
ALIGN 16
MEMSAVE   DB   512 DUP (0)
```

Збереження регістрів здійснюється командою

```
FXSAVE MEMSAVE,
```

а відновлення командою

```
FXRSTOR MEMSAVE
```

При необхідності значення зі змінної можна перенести у стек програми.

2.4. Скалярне множення двох векторів з використанням асемблера

Нижче наведений код функції обчислення скалярного множення двох векторів довільної довжини. Вирівнювати адреси розміщення векторів не потрібно (використовується операція переміщення невирівняних даних).

У основному циклі здійснюється одночасне множення чотирьох елементів векторів та накопичення чотирьох проміжних сум.

Якщо розмір вектора не кратний 4, після закінчення основного циклу передбачається множення елементів та складання залишку вектора (від 1 до 3 елементів).

```
float Scalar_prod   (float   *a,  float  *b,  int n)
{float sm;
      _asm  {
            pusha
```

```
            mov edi, a
            mov esi, b
            mov ecx, n
            xor ebx, ebx
            mov edx, ecx
            and edx, 3
            shr ecx, 2
            xorps xmm3, xmm3
        }
m1:     _asm
        {movups xmm0, [edi + ebx * 4]
        movups xmm1,   [esi + ebx * 4]
        add    ebx, 4
        mulps  xmm1, xmm0
        addps  xmm3, xmm1
loop m1
        movhlps xmm1, xmm3
        addps   xmm3, xmm1
        pshufd  xmm1, xmm3, 1
        addps   xmm3, xmm1
        movss sm, xmm3
        mov ecx, edx; edx
        }
m3:     _asm{       jcxz m2
                    fld dword ptr [edi + ebx * 4]
                    fld dword ptr [esi + ebx * 4]
                    fmul
                    dec ecx
                    fadd sm
```

```
                inc ebx
                fstp sm
                jmp m3
        }
m2:     _asm  popa
        return sm;
}
```

2.5. Порядок виконання роботи

Відповідно до варіанта завдання (табл. 2.6) реалізувати векторну обробку даних для матриці (з однаковим розміром матриць і векторів кратним чотирьом) з використанням розширень *SSE*2. У виразах α та β – константи, що вводяться з клавіатури, A та B – матриці (двовимірні масиви, розмірність кратна 4), x та y вектори.

Таблиця 2.6 –Варіанти завдань

Варіант	Формула для обчислення
1	$\max(\alpha Ax, \beta By)$
2	$\sqrt{\min(\alpha Ax, \beta By)}$
3	$\sqrt{1/(\alpha Ax - By)}$
4	$\min(Ax, \sqrt{\alpha By})$
5	$\sqrt{\min(\alpha Ax, By)}$
6	$\min(\sqrt{\alpha Ax}, \beta By)$
7	$\sqrt{(\alpha Ax)/(\beta By)}$
8	$\min(1/(\alpha Ax), \sqrt{By})$
9	$\max(Ax, 1/(By))$
10	$\sqrt{\alpha Ax + \max(By, Ax)}$

Операції з узяття мінімального, максимального значень і обчислення кореня квадратного і зворотного значення виконуються по елементах над відповідними аргументами. Введення даних і виведення результату реалізувати з використанням консольної програми на *C++*. Розрахунок результуючого вектора виконати у вигляді процедури з використанням асемблера.

Контрольні питання

1. Де знаходяться регістри *MMX* процесора?
2. Які типи даних обробляються за допомогою регістрів *MMX*?
3. Як виконуються операції в арифметиці з насиченням?
4. Які типи даних дозволяють обробляти регістри *XMM*?
5. У чому особливість векторних і скалярних операцій *SSE*?
6. Які вимоги висуваються до розміщення даних у пам'яті машини при обробленні за допомогою інструкцій *SSE*?
7. Яка директива асемблера забезпечує вирівнювання адреси розміщення даних в пам'яті обчислювача?
8. У якому випадку потрібно використовувати операцію збереження і відновлення значень регістрів *MMX* і *XMM*?
9. Який об'єм пам'яті потрібно для збереження стану обчислювача?
10. У чому відмінність операції *MOVAPD* від *MOVUPD*?
11. Яке використання джерела і приймача даних неприпустимо в операції переміщення даних?

3. ВИКОРИСТАННЯ ВБУДОВАНИХ ФУНКЦІЙ С ДЛЯ РОБОТИ З *SIMD*-РОЗШИРЕННЯМ

Мета роботи : вивчити набір функцій роботи з *SIMD*-розширенням архітектури x86, отримати навички розробки програм обробки векторних даних.

3.1. Розширення SSE процесора

Сучасні процесори мають додатковий модуль *SSE*(*Streaming SIMD Extensions*, потокове *SIMD* – розширення процесора) роботи з векторними даними.

*SSE*2 включає в архітектуру процесора вісім 128-бітових регістрів і набір інструкцій, працюючих із скалярними і упакованими (векторними) типами даних.

Перевага у продуктивності досягається в *SSE* за рахунок розпаралелювання обчислювального процесу між даними.

3.1.1. Оголошення змінних для використання у вбудованих функціях

Убудовувані функції використовують спеціальні 128-бітові дані.

Для цієї мети в заголовному файлі *emmintrin.h* оголошені типи: _ _m128, _ _m128i та _ _m128d, які є об'єднаннями.

Тип _ _m128 оголошений як

```
typedef union _ _declspec(intrin_type) _CRT_ALIGN(16) _ _m128 {
    float        m128_f32 [4];
    unsigned _ _int64      m128_u64 [2];
    _ _int8    m128_i8   [16];
    _ _int16   m128_i16  [8];
    _ _int32   m128_i32  [4];
    _ _int64   m128_i64  [2];
    unsigned _ _int8       m128_u8  [16];
    unsigned _ _int16      m128_u16 [8];
    unsigned _ _int32      m128_u32 [4];
 } _ _m128;
```

Типи _ _*m*128*i* та _ _*m*128*d* застосовуються для роботи з цілочисельними і двома дійсними числами у форматі подвійної точності.

Для найбільшої ефективності елементи таких типів даних мають бути вирівняні в пам’яті по відповідній межі. Адреси розміщення змінних типу _ _*m*128 мають бути кратні 16.

При необхідності вирівнювання розміщення даних у програмі може бути використана специфікація

_ _*declspec*(*align*(*N*)),

де *N* кратність адреси розміщення.

Наприклад, при оголошенні

_ _*declspec*(*align*(8)) *double R*[2];

масив *R* буде розміщений за адресою, яка кратна 8.

Статичні змінні і масиви компілятор вирівнює автоматично. Динамічні дані компілятор зазвичай вирівнює за адресою, яка кратна 4. Для виділення динамічної пам’яті з вирівнюванням використовується функція

*void *_mm_malloc*(*int size, int align*) ,

де *size* – об’єм пам’яті, що виділяється у байтах, *align* – межа вирівнювання.

Для звільнення пам’яті, виділеної за допомогою _mm_malloc, використовується функція

void _mm_free(*void *p*),

де *p* – покажчик області виділеної пам’яті.

Наприклад:

```
float *x;       // масив для обробки за допомогою інструкцій SSE
x=(float)_mm_malloc(100*sizeof(float),16);
...// обробка даних
_mm_free(x);
```

3.2. Вбудовані функції розширення *SSE*

Для роботи з *SSE* в *C/C++* є вбудовані функції. Для використання цих функцій у проект необхідно підключити заголовний файл *emmintrin.h*. Кожна

функція є вбудованою (*intrinsic*) послідовністю команд роботи з регістрами *SSE*.

Програми, в яких використовуються вбудовані функції, виконуються швидше, оскільки вони не породжують додаткове навантаження, пов'язане з викликом функцій і повернення з неї.

Для оголошення вбудованої функції використовується директива *#pragma intrinsic* (ім'я функції).

3.2.1. Арифметичні функції

Арифметичні функції забезпечують виконання операцій складання, віднімання, множення, ділення (табл. 3.1). Ці функції мають два аргументи – 128-розрядних змінних.

Таблиця 3.1 – Арифметичні функції

Функція	Операція
_mm_add_ss, _mm_add_sd	$r_0 = a_0 + b_0$, $r_i = a_i$, i=1÷N
_mm_add_ps ,_mm_add_pd	$r_i = a_i + b_i$, i=0÷N
_mm_sub_ss , _mm_sub_sd	$r_0 = a_0 – b_0$, $r_i = a_i$, i=1÷N
_mm_sub_ps , _mm_sub_pd	$r_i = a_i – b_i$, i=0÷N
_mm_mul_ss , _mm_mul_sd	$r_0 = a_0 \times b_0$, $r_i = a_i$, i=1÷N
_mm_mul_ps , _mm_mul_pd	$r_i = a_i \times b_i$, i=0÷N
_mm_div_ss , _mm_div_sd	$r_0 = a_0 / b_0$, $r_i = a_i$, i=1÷N
_mm_div_ps , _mm_div_pd	$r_i = a_i / b_i$, i=0÷N
_mm_sqrt_ss ,_mm_sqrt_sd	$r_0 = \sqrt{a_0}$, $r_i = a_i$, i=1÷N
_mm_sqrt_ps ,_mm_sqrt_pd	ri =√ai , i=0÷N
_mm_rcp_ss	$r_0 = 1/a_0$, $r_i = a_i$, i=1÷N
_mm_rcp_ps	$r_i = 1/a_i$, i=0÷N
_mm_rsqrt_ss	$r_0 = 1/a_0$, $r_i = a_i$, i=1÷N
_mm_rsqrt_ps	$r_i = 1/\sqrt{a_i}$, i=0÷N

Продовження табл. 3.1

Функція	Операція
_mm_min_ss ,*_mm_min_sd*	$r_0=\min(a_0,b_0)$, $r_i = a_i$, i=1÷N
_mm_min_ps ,*_mm_min_pd*	$ri=\min(a_i,b_i)$, i=0÷N
_mm_max_ss ,*_mm_max_sd*	$r_0=\min(a_0,b_0)$, $r_i = a_i$, i=1÷N
_mm_max_ps ,*_mm_max_pd*	$ri=\min(a_i,b_i)$, i=0÷N

Функції мають вигляд:

_ _*m*128 *_mm_XXXXs* (_ _*m*128 *a*, _ _*m*128 *b*)

для обробки дійсних даних одинарної точності і

_ _*m*128*d* *_mm_XXXXd* (_ _*m*128*d a*, _ _*m*128*d b*)

для обробки дійсних даних подвійної точності. *XXXX* – ім'я операції.

Функції обчислення квадратного кореня і знаходження зворотного значення мають один аргумент і мають вигляд:

_ _*m*128 *_mm_XXXXs* (_ _*m*128 *a*)

_ _*m*128*d* *_mm_XXXXd* (_ _*m*128*d a*)

для обробки дійсних даних одинарної і подвійної точності

3.2.2. Функції порівняння

У векторній формі порівнюються чотири дійсних значень параметра *a* з чотирма дійсними значеннями параметра *b*, і повертається 128-бітова маска. У скалярній формі порівнюються молодші значення параметрів, повертається 32-бітова маска, інші три старші значення копіюються з параметра *a*. Маска встановлюється в значення 0xffffffff для тих елементів, результат порівняння яких істина, і 0x0 в іншому випадку.

_ _*m*128 *_mm_cmpXXX*(_ _*m*128 *a* , _ _*m*128 *b*);

Набір функцій типу

int *_mm_comXXX*(_ _*m*128 *a* , _ _*m*128 *b*)

дозволяє порівнювати молодші елементи двох аргументів і повертати 1, якщо результат порівняння істина, інакше 0.

У табл. 3.2 наведені імена функцій порівняння.

Таблиця 3.2 – Функції порівняння

Функція	Операція
_mm_cmpeq_ss , _mm_cmpeq_sd	$r_0 = a_0 = = b_0$, $r_i = a_i$, i=1÷N
_mm_cmpeq_ps , _mm_cmpeq_pd	$r_i = a_i = = b_i$, i=0÷N
_mm_cmplt_ss, _mm_cmplt_sd	$r_0 = a_0 < b_0$, $r_i = a_i$, i=1÷N
_mm_cmplt_ps , _mm_cmplt_pd	$r_i = a_i < b_i$, i=0÷N
_mm_cmple_ss, _mm_cmple_sd	$r_0 = a_0 \leq b_0$, $r_i = a_i$, i=1÷N
_mm_cmple_ps, _mm_cmple_pd	$r_i = a_i \leq b_i$, i=0÷N
_mm_cmpgt_ss, _mm_cmpgt_sd	$r_0 = a_0 > b_0$, $r_i = a_i$, i=1÷N
_mm_cmpgt_ps, _mm_cmpgt_pd	$r_i = a_i > b_i$, i=0÷N
_mm_cmpge_ss, _mm_cmpge_sd	$r_0 = a_0 \geq b_0$, $r_i = a_i$, i=1÷N
_mm_cmpge_ps, _mm_cmpge_pd	$r_i = a_i \geq b_i$, i=0÷N
_mm_cmpneq_ss, _mm_cmpneq_sd	$r_0 = a_0 \neq b_0$, $r_i = a_i$, i=1÷N
_mm_cmpneq_ps, _mm_cmpneq_pd	$r_i = a_i \neq b_i$, i=0÷N
_mm_cmpord_ss, _mm_cmpord_sd	$r_0 = a_0$ порівняно с b_0
_mm_cmpord_ps, _mm_cmpord_pd	$r_i = a_i$ порівняно с b_i i=1÷N
_mm_cmpunord_ss, _mm_cmpunord_sd	$r_0 = a_0$ не порівняно с b_0
_mm_cmpunord_ps, _mm_cmpunord_pd	$r_i = a_i$ не порівняно с b_i i=1÷N
_mm_comieq_ss, _mm_comieq_sd	1 якщо $a_0 = = b_0$, інакше 0
_mm_comilt_ss, _mm_comilt_sd	1 якщо $a_0 < b_0$, інакше 0
_mm_comile_ss, _mm_comile_sd	1 якщо $a_0 \leq$ be, інакше 0
_mm_comigt_ss, _mm_comigt_sd	1 якщо $a_0 > b_0$, інакше 0
_mm_comige_ss, _mm_comige_sd	1 якщо $a_0 \geq b_0$, інакше 0
_mm_comineq_ss, _mm_comineq_sd	1 якщо $a_0 \neq b_0$, інакше 0

3.2.3. Функції ініціалізації

Функції ініціалізації забезпечують запис дійсних значень, що зберігаються в поодинокій змінній або масиві, у змінні типу _ _*m*128.

Функції ініціалізації мають один аргумент – покажчик на область пам'яті, в якій зберігається одно або декілька дійсних чисел одинарної або подвійної точності. Усі функції повертають тип даних _ _*m*128 та мають вигляд:

_ _*m*128 _*mm_loadXXs* (*float * p*);

_ _*m*128*d* _*mm_loadXXd* (*double * p*);

для обробки дійсних даних одинарної і подвійної точності.

У табл. 3.3 наведені імена функцій ініціалізації.

Для запису нульових значень можна використати функції

_ _m128 _mm_setzero_ps ( )

_ _m128d _mm_setzero_pd ( ) ,

які записують нульові значення в усі позиції.

Таблиця 3.3 – Ініціалізація пам'яті

Функція	Операція
_mm_load_ss, _mm_load_sd	Завантажити молодше значення та інші очистити
_mm_load1_ps, _mm_load1_pd	Завантажити одно значення в усі позиції
_mm_load_ps,_mm_load_pd	Завантажити усі позиції, джерело вирівняне
_mm_loadu_ps,_mm_loadu_pd	Завантажити усі позиції, джерело не вирівняне
_mm_loadr_ps,_mm_loadr_pd	Завантажити значення в зворотному порядку

3.2.4. Функції збереження

Функції збереження дозволяють переписувати значення, що зберігаються в змінних типу _ _m128 в змінні або масиви дійсного типу. Функція має вигляд:

void _mm_storeXXs (*float *p, _ _m*128 *a*);

void _mm_storeXXd (*double *p, _ _m*128*d a*);

У таблиці 3.4 представлені операції збереження даних.

3.2.1. Функції перестановки елементів

Для роботи з дійсними числами одинарної точності є функції перестановки *_ mm _ shuffle _ ps*, *_ mm _ movehl _ ps*, *_ mm _ movelh _ ps*.

Таблиця 3.4 – Функції збереження

Функція	Операція
_mm_store_ss,_mm_store_sd	Записати значення у змінну
_mm_store_ps,_mm_store_pd	Записати усі елементи в масив вирівняний за адресою
_mm_storeu_ps,_mm_storeu_pd	Записати усі елементи в масив не вирівняний за адресою
_mm_storer_ps,_mm_storer_pd	Записати усі значення у зворотному порядку в масив вирівняний за адресою

Функція перестановки елементів

_ _*m*128 _ *mm_shuffle*_ps(_ _*m*128 *a*, _ _*m*128 *b*, *int j*)

дозволяє сформувати нове значення типу з двох інших. Третій параметр є маскою порядку завантаження результату з операндів джерел. Значення аргументу *j* може бути сформоване за допомогою макросу *_ MM _ SHUFFLE*(*z*, *y*, *x*, *w*), де *z* та *y* –два індекси елементів другого параметра функції, а *x* та *w* – індекси елементів першого параметра. Значення *z, y, x, w* знаходяться в межах від 0 до 3. Формування результату перестановки елементів показане на рис. 3.1.

Функція _ _*m*128 *_mm_movehl_ps*(_ _*m*128 *a*, _ _*m*128 *b*)

здійснює копіювання старших половин двох змінних (рис. 3.2).

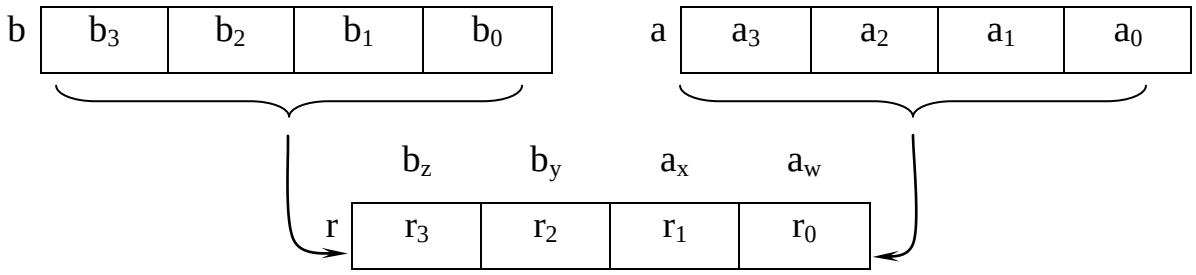

Рисунок 3.1 – Формування результату функції *_mm_shuffle_ps*

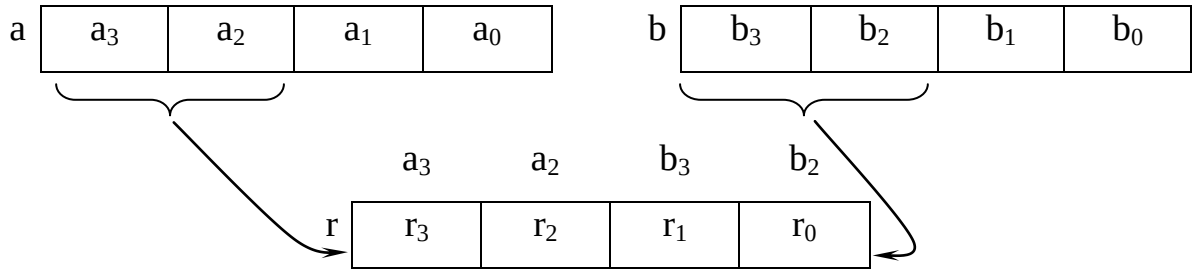

Рисунок 3.2 – Формування старших половин результату

Функція _ _*m*128 *_mm_movelh_ps* (_ _*m*128 *a*, _ _m128 *b*) здійснює копіювання молодшої половини двох змінних (рис. 3.3).

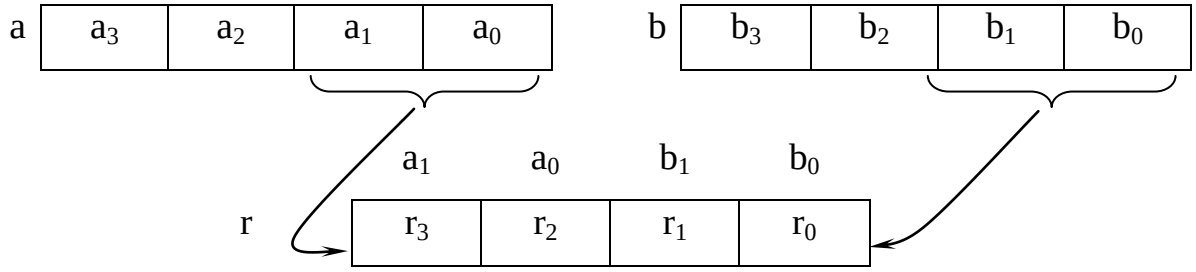

Рисунок 3.3 – Формування молодшої половини результату

3.2.2. Функції перетворення типів

Функції перетворення типів (табл. 3.5) змінюють форму представлення числових даних.

При перетворенні дійсних даних в цілочисельну форма подання функції *_mm_cvtss_si32* та *_mm_cvtps_pi32* здійснюють округлення, а функції *_m_cvttss_si32* та *_mm_cvttps_pi32* відкидання дробової частини.

Таблиця 3.5 – Функції перетворення типів

Функція	Перетворення
int _mm_cvtss_si32 (_ _m128 a);	*float* в *int* (32 біт)
_ _m64 _mm_cvtps_pi32(_ _m128 a)	два молодших *float* у два *int* (32 біт)
int _mm_cvttss_si32 (_ _m128 a)	*float* в *int* (32 біт)
_ _m64 _mm_cvttps_pi32 (_ _m128 a);	два *float* в два *int* (32 біт)
_ _m128 _mm_cvtsi32_ss (_ _m128 a , int b);	*int* (32 біт) во *float*
_ _m128 _mm_cvtpi32_ps (_ _m128 a ,_ _m64 b);	два *int* (32 біт) в два *float*
_ _m128 _mm_cvtpi16_ps (_ _m64 a);	чотири *short* (16 біт) в чотири *float*
_ _m128 _mm_cvtpu16_ps (_ _m64 a);	чотири *short*(без знаку) у чотири *float*
_ _m128_mm_cvtpi8_ps (_ _m64 a);	чотири *byte* (8 біт) у чотири *float*
_ _m128 _mm_cvtpu8_ps (_ _m64 a);	чотири молодших *byte* (8 біт, без знаку) в чотири *float*
_ _m128 _mm_cvtpi32x2_ps(_ _m64 a, _ _m64 b);	дві пари *int* (два по 32 біт) в чотири *float*
_ _m64 _mm_cvtps_pi16 (_ _m128 a);	чотири *float* в чотири *short* (16 біт)
_ _m64 _mm_cvtps_pi8 (_ _m128 a);	чотири *float* в чотири *byte* (8 біт)

3.3. Використання вбудованих функцій SSE у програмі на мові Сі

Нижче наведений текст програми обчислення скалярного множення двох векторів.

Головна функція формує два вектори. Елементи першого вектора заповнюються числами 10 * *i* / *N*, де *i* – номер елементу, *N* – загальне число елементів. Другий вектор заповнюється цими числами у зворотному порядку.

Масиви для зберігання векторів виділяються динамічно з урахуванням вирівнювання на межу, кратну 16, за допомогою функції *_mm_malloc*. Це забезпечує швидший спосіб звернення до елементів масиву.

Функція *Scalar_Cpp* обчислює скалярне множення з використанням засобів *C*, а *Scalar_SSE* – з використанням вбудованих *SSE* функцій.

Для виміру часу виконання використовувалася функція *QueryPerformanceCounter*. При малих розмірах векторів час виконання обох функцій мало чим відрізняється. Але вже при розмірності *N*=1000 функція *Scalar _ SSE* працює у три рази швидше, ніж *Scalar_Cpp*.

Ще однією особливістю *Scalar_SSE* є підвищена точність при великих розмірах векторів (близько 1 000 000 елементів). Це пов'язано з формуванням ситуації зникнення порядку. Оскільки в *Scalar_SSE* набуття проміжних значень розбите на чотири частини, ситуація зникнення порядку настає при більших значеннях *N*, чим в *Scalar_Cpp*.

```
#include <stdio.h> // скалярне множення векторів
#include <emmintrin.h>
#include <time.h>
#include <windows.h>  //
#define N 100000
float Scalar_Cpp (int n, float *x, float *y)
{       float s = 0;
        for (int i = 0; i<n; i++)
 s += x[i] * y[i];
        return s;
}
float Scalar_SSE (int n, float *x, float *y,)
{       float sum;
        _ _m128 *xx, *yy, s;
        xx = (_ _m128 *)x;
        yy = (_ _m128 *)y;
        s = _mm_setzero_ps();
        for (int i = 0; i<n / 4; i++)
        {
        s = _mm_add_ps(s, _mm_mul_ps(xx[i], yy[i]));
        }
```

```
        s = _mm_add_ps(s, _mm_movehl_ps(s, s));
                s = _mm_add_ss(s, _mm_shuffle_ps(s, s, 1));
        _mm_store_ss(&sum, s);
        return sum;
}
int main( )
{
float *x, *y, s;
        LARGE_INTEGER b_start, b_stop, b_time;
        b_start.QuadPart = b_stop.QuadPart = b_time.QuadPart= 0;
        double time, pi;

        x = (float *)_mm_malloc(N*sizeof(float), 16);
        y = (float *)_mm_malloc(N*sizeof(float), 16);
        for (int i = 0; i<N; i++)
        {
                x[i] = 10 * i / N;
                y[i] = 10 * (N – i – 1) / N;
        }
        QueryPerformanceCounter(&b_start);
                s = Scalar_Cpp (N, x, y);
        QueryPerformanceCounter(&b_stop);
        b_time.QuadPart = b_stop.QuadPart – b_start.QuadPart;
printf("Result: %f time %lf\n", s,(double)(b_time.QuadPart) );
        QueryPerformanceCounter(&b_start);
                s = Scalar_SSE (N, x, y);
        QueryPerformanceCounter(&b_stop);
        b_time.QuadPart = b_stop.QuadPart – b_start.QuadPart;
printf("Result: %f time %lf\n", s,  (double)(b_time.QuadPart));
```

```
    _mm_free(x);
    _mm_free(y);
    getchar();
    return 0;
}
```

3.4. Налаштування середовища програмування *Visual Studio* для використання *SSE* команд

Для завдання режиму використання інструкцій *SSE* необхідно встановити параметр вибору архітектури (/*arch*). Установлення параметра здійснюється у властивостях проекту. У розділі налаштування транслятора *C/C++* в пункті "Створення коду" для параметра "Включити розширений набір інструкцій" можна вибрати необхідний набір інструкцій для генератора коду (*SSE, SSE*2, *AVX, AVX*2 або *IA*32). При установленні режиму використання інструкцій *SSE* і *SSE*2 транслятор для скалярних обчислень з плаваючою комою використовуватиме розширення *SSE*/*SSE*2 замість співпроцесора x87, якщо така заміна забезпечує більш високу швидкодію програми. Крім того, використовуючи режим *SSE*2, транслятор може використати інструкції *SSE*2 для деяких операцій над 64-розрядними цілими числами.

3.5. Порядок виконання роботи

Відповідно до варіанта завдання необхідно реалізувати векторні операції для матриці (з однаковим розміром матриць, кратним чотирьом) з використанням розширень *SSE*2 (табл.3.6).

Передбачається, що *A, B, C* та *D* – матриці, а усі операції виконуються по елементам. Введення даних і виведення результату реалізувати з використанням консольної програми на мові *C* . Розрахунок результуючого вектора виконати двома варіантами:

- з використанням вбудованих функцій *SSE*;
- з використанням мови *C* .

Отримати графік залежності часу виконання програми від розмірності масивів і векторів.

Таблиця 3.6 – Варіанти завдань

Варіант	Формула
1	$\mathrm{A} + \min(B, C)/D$
2	$\max((\mathrm{A} + \mathrm{B}), (\mathrm{D} - \mathrm{C}))$
3	$\sqrt{(A * B - C/D)}$
4	$A/\sqrt{A * \min(B, C)}$
5	$\sqrt{A - B} * \sqrt{D/C}$
6	$\mathrm{A} + \min(B * C, D)$
7	$\max((\mathrm{A} + \mathrm{B}) * (\mathrm{D} - \mathrm{C}))$
8	$\min(\sqrt{(A * B)}, C) + D$
9	$(A + B)/\sqrt{D - C}$
10	$\max(\sqrt{A - B}, \sqrt{D/C})$

Контрольні питання

1. Який файл необхідно підключити для роботи з вбудованими функціями *SSE*?
2. Які типи даних обробляються вбудованими функціями *SSE*?
3. Як здійснюється вирівнювання статичних даних в *C*?
4. Як забезпечити вирівнювання даних при динамічному виділенні пам’яті?
5. Як об’явити призначену для користувача вбудовану функцію?
6. Які арифметичні операції над скалярними і векторними даними реалізовані у вигляді вбудованих функцій?

СПИСОК ЛІТЕРАТУРИ

1. Юров В.И. Assembler: Специальный справочник. 2–е изд./ В. И. Юров – СПб: Питер, 2004. – 412 с.:ил.

2. Воеводин В. В. Параллельные вычисления / В. В.Воеводин, Вл. В Воеводин – СПб: БХВ–Петербург, 2002. – 608 с.

3. Злобін Г.Г. Архітектура та апаратне забезпечення ПЕОМ : навч.посіб./ Г. Г. Злобін –К.: Каравела, 2006.–304 с

СОДЕРЖАНИЕ

Навчальне видання

Методичні вказівки

до лабораторних робіт

“ Використання SIMD розширення обчислювача при програмуванні в середовищі VISUAL STUDIO”

з дисципліни “ Архітектура обчислювальних систем ”

для студентів спеціальностей 113 – Прикладна математика, 122 – Комп’ютерні науки та інформаційні технології, 124 – Системний аналіз, 186 – Видавництво та поліграфія

Укладачі:

КОЖИН Юрій Миколайович

МАЛИХ Олег Миколайович

ПРОКОПЕНКОВ Володимир Пилипович

Відповідальний за випуск О.С. Куценко

Роботу до друку рекомендував М.І. Безменов

Редактор О.І. Шпільова

План 2017, поз. 133

Підп. до друку.	Формат 60x84 1/16.	Папір офсетний.
Riso-друк.	Гарнітура Таймс.	Ум. друк. арк. 2,1.
Наклад 25 прим.	Зам. №	Ціна договірна.

Видавничий центр НТУ “ХПІ”

м. Харків, 61002, вул. Кирпичова, 21

Свідоцтво суб’єкта видавничої справи ДК №3657 від 24.12.2009 р